Nate Stewart
Dr. Jeffrey Jones (TuTh 12:45pm – 2:05pm)
04-11-16
Lab 4 – Matrix Math using CUDA

Environment:

All test results presented in this report were run on the Oakley cluster using and interactive node with 1 node, 1 gpu, and a walltime of 59 minutes. All CUDA compilation occurred using the default version (6.5.14). The three programs were written using CUDA C and compiled using the following command: nvcc -O -o <output_name> <input_file>. The three programs should be run with the following command: 'time ./lab4pX' where X is 1, 2, or 3. Unfortunately, the only timing method used was unix time because ctime and clock provided very inconsistent numbers which made the Gflops calculations difficult.

Results:

Detailed run time and Gflops information is available in the table below, Figure 1. Only the runs with the highest Gflops are included in the table (except for part 3 which required several different runs and no serial run). Across the board, parallelizing the programs resulted in faster runtimes and higher Gflops.

Program	Num Blocks	Num Threads	Real Time	Number of Flop	Gflops
Part 1 Serial	0	1	2.347	1680588800	0.7160582872
Part 1 Parallel	64	64	2.044	1681817600	0.822807045
Part 2 Serial	0	1	1564.031	274911461376	0.1757711077
Part 2 Parallel	524288	32	9.982	275062456320	27.555846155
Part 3 Parallel	16384	32	0.103	5239808	0.0508719223
Part 3 Parallel	1	32	0.164	3142720	0.0191629268
Part 3 Parallel	1	1024	0.114	3143680	0.0275761404
Part 3 Parallel	2	1024	0.111	3148800	0.0283675676
Part 3 Parallel	16	1024	0.109	3143680	0.0288411009

Figure 1: Completion metrics for the three different parts

The results of parallelizing the first part are less than ideal. The number of floating point operations increased by a very marginal amount after parallelization, and the execution time decreased by .3 seconds. This resulted in an overall performance boost of about 0.115 Gflops. Although the performance increased slightly, the accuracy of the results became indeterminate. CUDA is not good at parallelizing loop dependencies due to its lack of synchronization, and therefore, part 1 should not be parallelized with CUDA. There are two loop dependencies in the program so using CUDA, even with 1 block and 1 thread (see Piazza discussion board), results in inconsistent results depending on which row and column is computed first.

The results of parallelizing the second part were much better. I was able to decrease the execution from 1564 seconds (26 minutes and 4 seconds) to a little under 10 seconds. I was unable to decrease the number of floating point operations, but my thread structure took care of the speedup. This increased the number of Gflops from 0.18 to 27.56. To accomplish this, I used a 2D grid structure of 4096 x 128 blocks with each block having 32 threads. This allowed for each of the 16777216 threads to compute a single row of 4096 elements. By using thread pointers to the two operands of the multiplicaion and a temporary partial sum variable, I was able to minimize the number of address loads. I ran the parallel program on a large number of grid/block structure combinations and found that 32/64 threads provided the best performance on the GPUs hosted on OSC.

The results of the third part were nearly identical. The number of floating point operations among all of the part three programs was also nearly identical (except for my custom implementation which had about 2.1 million more operations). Using 1 block with 32 threads yielded the worst results (~0.019 Gflops) due to how much work was being computed by each thread. Using 1 block with 1024 threads and 2 blocks with 1024 threads each both resulted in ~0.28 Gflops. Running the program with 16 blocks of 1024 threads each resulted in a marginal performance improvement (< 0.0005 Gflops better than 2 blocks of 1024 threads each). These results were not unexpected since decreasing the amount of work per thread, up to a limit, will increase the performance. That being said, none of the implementations requested by the writeup performed better than my implementation of 16384 blocks of 32 threads each. Using this implementation, each thread computed a single transposition which resulted in ~0.051 Gflops and a runtime of 0.103 seconds.

Conclusion:

I took away two major ideas from this CUDA lab. First and most important, do not use CUDA to implement any programs that have loop dependencies that cannot be reworked. This was revealed during my implementation of the first lab where I was unable to get identical results as the serial program, even with using 1 block and 1 thread (see Piazza post for more information). Lab 2 and 3 were very parallelizable due to there being no loop dependencies and a fine line between operations. Second, keep the number of threads per block small. If you need more threads, add more blocks. This will prevent bottlenecking on the number of registers in use per block and allow all important operations to be completed in one clock cycle. In conclusion, CUDA is very powerful, but also very different from what I'm used to. So although there is a steep learning curve, CUDA is definitely worth learning.